SPI LLD User Guide

TABLE OF CONTENTS

1 Introduction

The SPI (Serial Peripheral Interface) driver is a generic driver which supports following two SPI peripherals: MCSPI (Multichannel SPI) and QSPI (Quad SPI).

MCSPI is a generic, full-duplex driver that transmits and receives data on a SPI bus. QSPI is variant of SPI which will have 4 receive data lines and can be configured to receive either using single, dual or quad data lines.

2 Key Features

- Operating modes
    - Blocking(interrupt or Non interrupt)
    - Callback mode(interrupt)

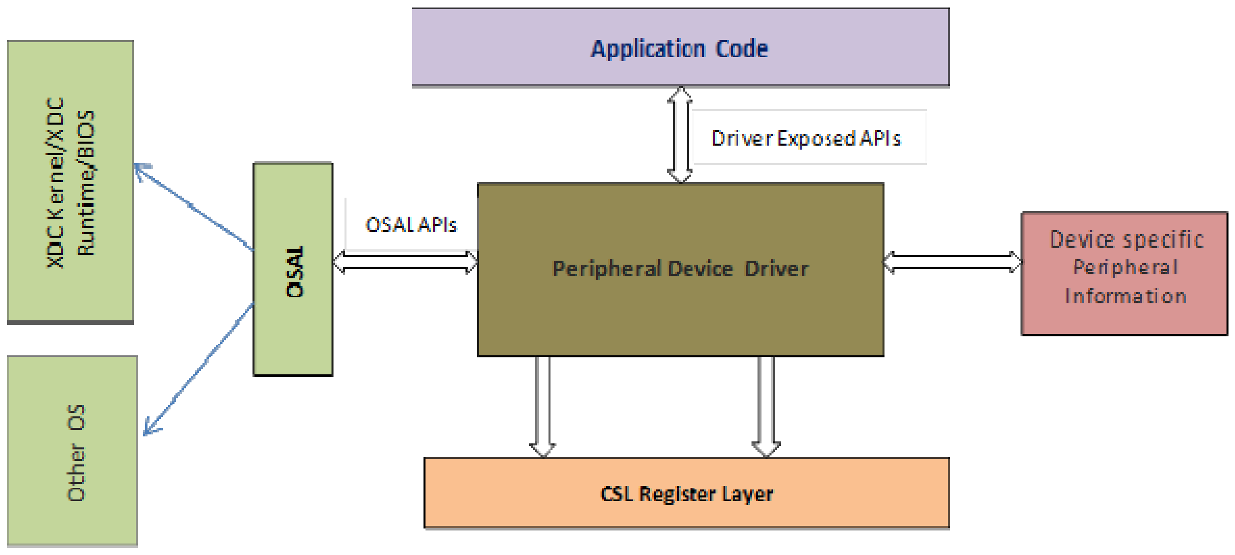
3 Hardware Support (EVM/SoCs)

Board	**SoC**	**Cores**
AM572x IDK EVM	AM572x	A15 & C66X
AM572x GP EVM	AM572x	A15 & C66X

4 Design Overview

The SPI driver provides a well-defined API layer which allows applications to use the SPI peripheral to send and receive data.

The below figure which shows the SPI Driver architecture.



The figure illustrates the following key components:-

4.1 Peripheral device driver

This is the core SPI device driver. The device driver exposes a set of well-defined APIs which are used by the application layer. The driver also exposes a set of well-defined OS abstraction APIs which will ensure that the driver is OS independent and portable. The driver uses the CSL register layer for MMR accesses.

4.2 Device specific module layer

This layer implements a well-defined interface which allows the core SPI device driver to be ported to any device which has the same SPI IP block. This layer may change for every device.

4.3 Application Code

This is the user of the driver and its interface through the well-defined APIs set. Application uses the driver APIs to send and receive data via the SPI peripheral.

4.4 OSAL

The driver is OS independent and exposes all the operating system callouts via this OSAL layer.

4.5 CSL Register Layer

The SPI driver uses the CSL SPI functional layer to program the device IP by accessing the MMR (Memory Mapped Registers).

5 Modes of Operation

SPI driver provides the following modes of operations.

5.1 SPI_MODE_BLOCKING

By default, the SPI driver operates in blocking mode. In blocking mode, a Task's code execution is blocked until a SPI transaction has completed. This ensures that only one SPI transaction operates at a given time.

This mode will support both interrupt or non-interrupt based blocking modes.

5.2 SPI_MODE_CALLBACK

In callback mode, a SPI transaction functions asynchronously, which means that it does not block code execution. After a SPI transaction has been completed, the SPI driver calls a user-provided hook function.

Only interrupt based callback mode is supported. Callback mode is not supported in case of non-interrupt use cases.

6 Driver Configuration

6.1 Board Specific Configuration

All the board specific configurations like enabling the clock and pin-mux of SPI pins should be performed before calling any of the driver APIs. Once the board specific configuration is done then the driver API SPI_init () should be called to initialize the SPI driver.

6.2 SPI Configuration Structure

The SPI_soc.c file contains the declaration of the SPI_config structure. This structure must be provided to the SPI driver. It must be initialized before the SPI_init () function is called and cannot be changed afterwards. For details about the individual fields of this structure, see the Doxygen help by opening \docs\doxygen\html\index.html.

6.2.1 Support of MCSPI and QSPI:

Since two peripherals (MCSPI and QSPI) are supported using this single driver. The SoC specific attributes corresponding to the both peripherals will be combined and will be placed in the file “SPI_soc.c”.

First all the MCSPI related hardware attributes will be defined for all the instances and these will be followed by QSPI hardware attributes. So if user has to access the hardware attributes corresponding to QSPI then while opening the driver for QSPI use case, user has to add the fixed offset to instance parameter, so that it will correctly fetch QSPI data from the SPI_config structure.

6.3 APIs

In order to use the SPI module APIs, the SPI.h header file should be included in an application as follows:

#include <ti/drv/spi/SPI.h>

The following are the SPI APIs:

- **SPI_init ()** initializes the I2C module.
- **SPI_Params_init ()** initializes an I2C_Params data structure. It defaults to Blocking mode.
- **SPI_open()** initializes a given I2C peripheral.
- **SPI_close()** deinitializes a given I2C peripheral.
- **SPI_transfer()** handles the I2C transfer.
- **SPI_transferCancel ()** Function to cancel the SPI transaction
- **SPI_control ()** Performs implementation specific features on a given instance of the peripheral.

6.4 Usage

The application needs to supply the following structures in order to set up the framework for the driver:

- **SPI_Params** specifies the transfer mode and any callback function to be used.
- **SPI_Transaction** specifies details about a transfer to be performed.
- **SPI_Callback** specifies a function to be used if you are using callback mode.

6.5 API Calling Sequence

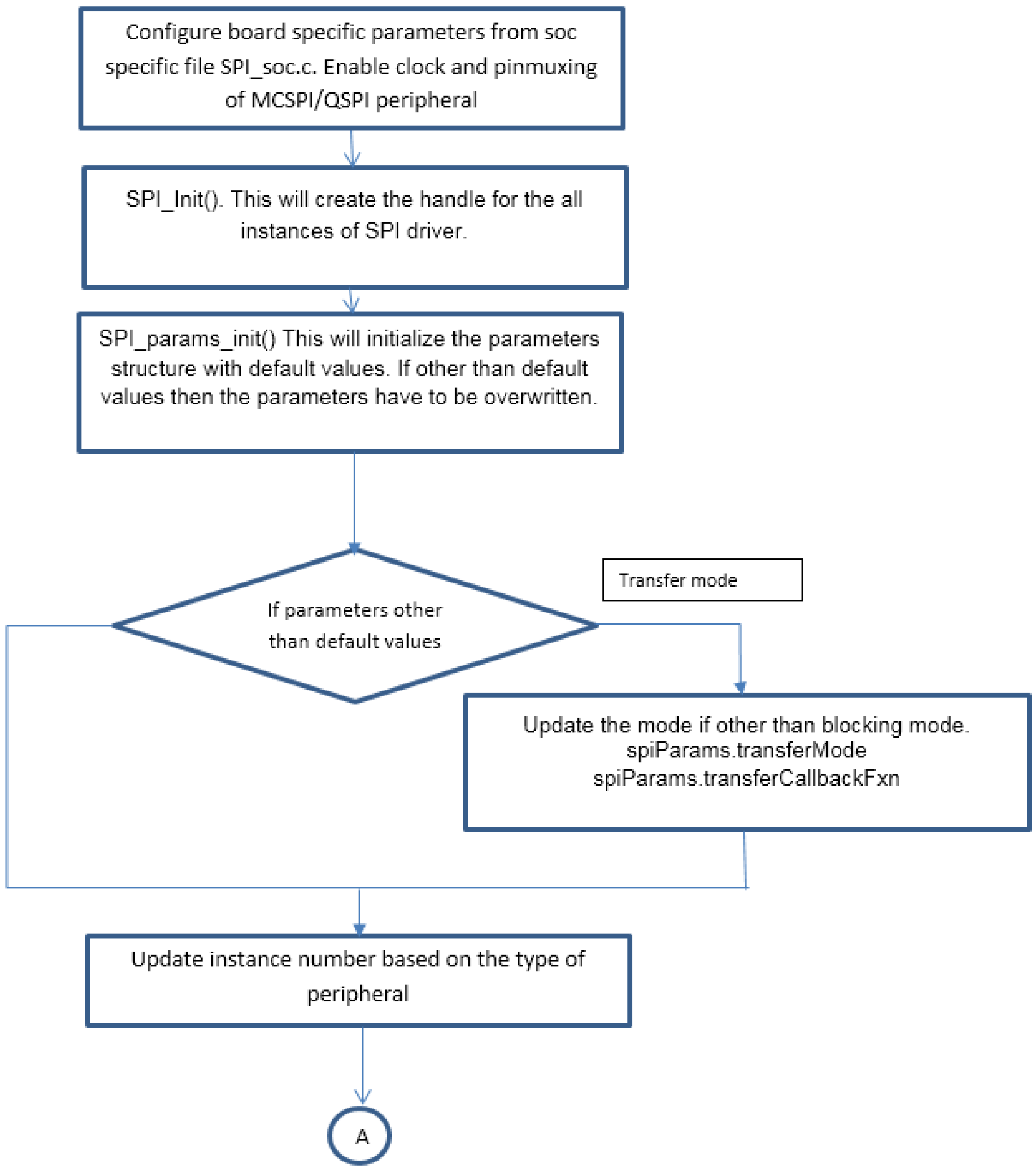
The below sequence indicates the calling sequence of SPI driver APIs for a use case of write transaction in blocking mode:

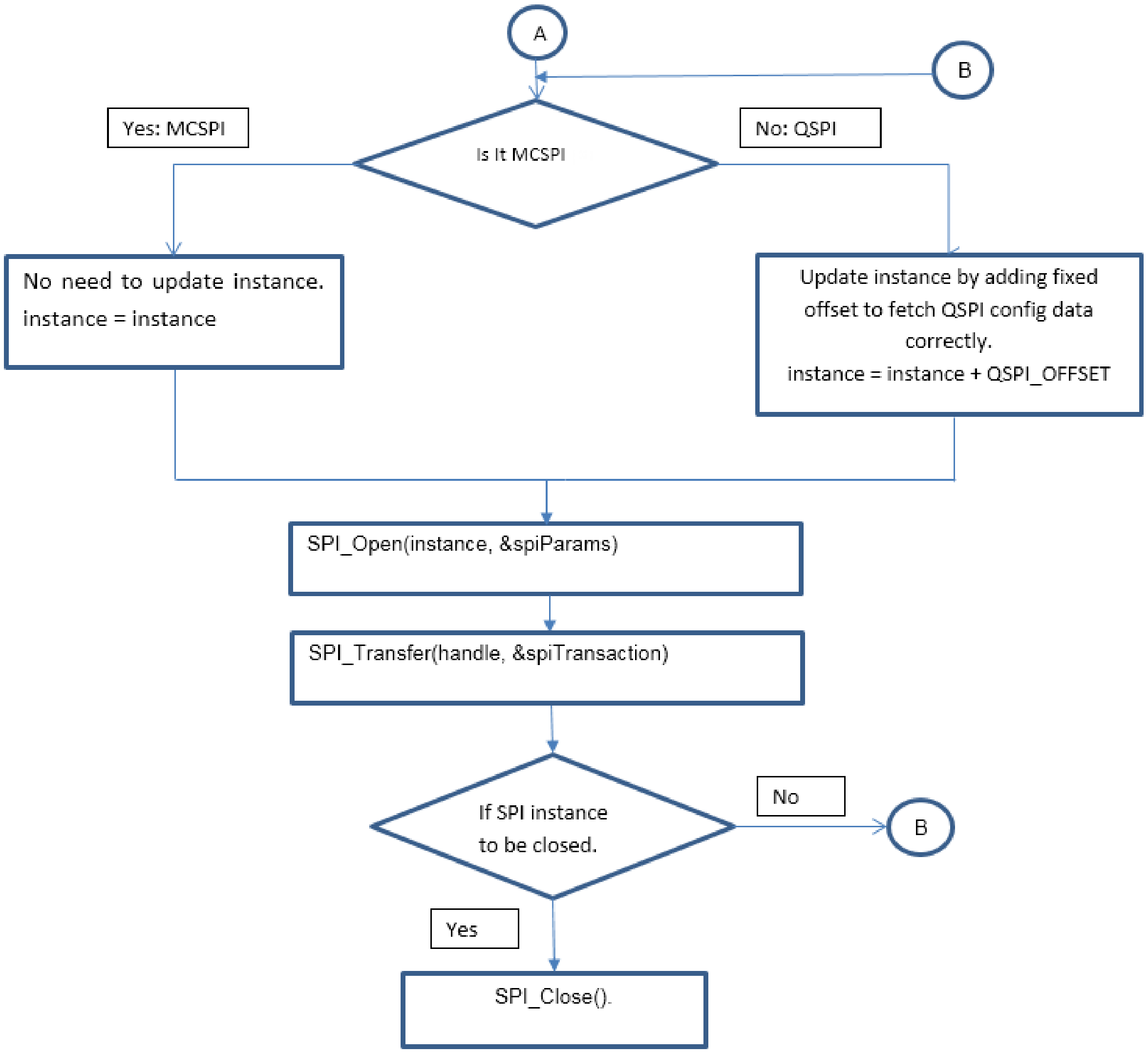
```
SPI_Handle spi;
UInt peripheralNum = 0;
SPI_Params spiParams;
SPI_Transaction spiTransaction;
uint8_t transmitBuffer[n];
uint8_t receiveBuffer[n];
bool transferOK;

SPI_Params_init(&spiParams);
spiParams.transferMode = SPI_MODE_BLOCKING;
spiParams.transferCallbackFxn = NULL;
```

```
spi = SPI_open(peripheralNum, &spiParams);
if (spi == NULL) {
   /* Error opening SPI */
}

spiTransaction.count = n;    /* Transfer Length */
spiTransaction. txBuf = transmitBuffer; /* Buffer to be written */
spiTransaction.rxBuf = receiveBuffer;  /* Buffet holding the received data */


transferOK = SPI_transfer(spi, &spiTransaction); /* Perform SPI transfer */
if (!transferOK) {
   /* SPI transaction failed */
}
```

6.6 Flow chart

API flow path of SPI driver is as shown in fig.


Configure board specific parameters from soc specific file SPI_soc.c. Enable clock and pinmuxing of MCSPI/QSPI peripheral
SPI_Init(). This will create the handle for the all instances of SPI driver.
SPI_params_init() This will initialize the parameters structure with default values. If other than default values then the parameters have to be overwritten.
If parameters other than default values
Transfer mode
Update the mode if other than blocking mode. spiParams.transferMode spiParams.transferCallbackFxn
Update instance number based on the type of peripheral
A


A
B
Yes: MCSPI
Is It MCSPI
No: QSPI
No need to update instance.
instance = instance
Update instance by adding fixed offset to fetch QSPI config data correctly.
instance = instance + QSPI_OFFSET
SPI_Open(instance, &spiParams)
SPI_Transfer(handle, &spiTransaction)
If SPI instance to be closed.
No
B
Yes
SPI_Close().

7 Examples

Following are the examples of supported for the SPI Driver

7.1 MCSPI

7.1.1 Serializer test:

This application will read the input data generated from the industrial input module. The data will be read through MCSPI interface. This application will read one byte of data from the industrial input module. To generate the data from the industrial input module, first load signal has to be asserted using the gpio pins.

On the IDK EVM, in the header J37, short the pins 1 and 2. This should generate an input "0x01" and the same should be read through the MCSPI interface.

The received data will vary based on the shorting of pins:

No pins are shorted	: received data - 0x00
Pins 1 and 2 shorted	: received data - 0x01
Pins 3 and 4 shorted	: received data - 0x02
Pins 5 and 6 shorted	: received data - 0x04
Pins 7 and 8 shorted	: received data - 0x08
Pins 9 and 10 shorted	: received data - 0x10
Pins 11 and 12 shorted	: received data - 0x20
Pins 13 and 14 shorted	: received data - 0x40
Pins 15 and 16 shorted	: received data - 0x80

7.2 QSPI

7.2.1 Flash read write

This example will write known data of fixed length to flash and reads the data from flash and verifies the read data for correctness.

7.3 Building the Examples

Following are list of SPI projects which will reside the following location “packages/MyExampleProjects”

MCSPI_BasicExample_AM571X_armExampleProject
MCSPI_BasicExample_AM571X_c66xExampleProject
MCSPI_BasicExample_AM572X_armExampleProject

MCSPI_BasicExample_AM572X_c66xExampleProject
QSPI_BasicExample_AM571X_armExampleProject
QSPI_BasicExample_AM571X_c66xExampleProject
QSPI_BasicExample_AM572X_armExampleProject
QSPI_BasicExample_AM572X_c66xExampleProject

These projects have to be imported in CCS and have to be built. The ".out" files corresponding to each project will be generated after successfully compiling the projects.

Interrupt/Non-Interrupt Modes:
The example projects have to be recompiled to support interrupt and non-interrupt use cases

Following parameter have to be updated in I2C_Soc.c file and the application have to be recompiled.

- Structure: spiInitCfg
- Parameter: enableIntr
  - True: interrupt
  - False: Non interrupt

7.4 Running the examples

The ".out" have to be loaded and executed. Then on the CCS console the result of the project execution will be displayed. If the project is executed successfully, then it will print "PASS" else will print "Data Mismatch".

7.5 Supported platforms:

AM572x IDK EVM
AM571x IDK EVM

8 Test

8.1 MCSPI

This test application will read the input data generated from the industrial input module. The data will be read through MCSPI interface. This application will read one byte of data from the industrial input module. To generate the data from the industrial input module, first load signal has to be asserted using the gpio pins.

On the IDK EVM, in the header J37, short the pins 1 and 2. This should generate an input "0x01" and the same should be read through the MCSPI interface.

The received data will vary based on the shorting of pins:

No pins are shorted	: received data - 0x00
Pins 1 and 2 shorted	: received data - 0x01
Pins 3 and 4 shorted	: received data - 0x02
Pins 5 and 6 shorted	: received data - 0x04
Pins 7 and 8 shorted	: received data - 0x08
Pins 9 and 10 shorted	: received data - 0x10
Pins 11 and 12 shorted	: received data - 0x20
Pins 13 and 14 shorted	: received data - 0x40
Pins 15 and 16 shorted	: received data - 0x80

8.2 QSPI

This test application will write known data of fixed length to flash and reads the same data and verifies the read data for correctness. This test application will test the QSPI driver in different modes (config and memory map modes). This test application will also test reception in different modes like single, dual and quad line reception.

8.3 Building the Examples

Following are list of SPI projects which will reside the following location "packages/MyExampleProjects"

MCSPI_BasicExample_AM571X_armTestProject
MCSPI_BasicExample_AM571X_c66xTestProject
MCSPI_BasicExample_AM572X_armTestProject
MCSPI_BasicExample_AM572X_c66xTestProject
QSPI_BasicExample_AM571X_armTestProject
QSPI_BasicExample_AM571X_c66xTestProject
QSPI_BasicExample_AM572X_armTestProject
QSPI_BasicExample_AM572X_c66xTestProject

These projects have to be imported in CCS and have to be built. The ".out" files corresponding to each project will be generated after successfully compiling the projects.

Interrupt/Non-Interrupt Modes:
The example projects have to be recompiled to support interrupt and non-interrupt use cases

Following parameter have to be updated in I2C_Soc.c file and the application have to be recompiled.

- Structure: spiInitCfg
- Parameter: enableIntr
  - True: interrupt
  - False: Non interrupt

8.4 Running the examples

The ".out" have to be loaded and executed. Then on the CCS console the result of the project execution will be displayed. If the project is executed successfully, then it will print "PASS" else will print "Data Mismatch".

8.5 Supported platforms:

AM572x IDK EVM
AM571x IDK EVM

9 Migration Guide

The driver supports multiple SoCs, Cores and different IP versions. Different IP versions are supported using function pointer based approach, whereas high lever driver APIs will remain same and these driver APIs will call the corresponding the correct version of IP specific implementation APIs using function pointers. This function pointer table will be fixed for each instance of the peripheral and will be defined in the main config structure, which resides in soc specific file "SPI_soc.c".

Users who are using the low level APIs(Device abstraction APIs: which perform hardware register read/write) have to use the high level APIs which are described in the section 6.3.

10 Benchmarking

Code size for library in bytes:

Initialized data	:	132
Code	:	12480